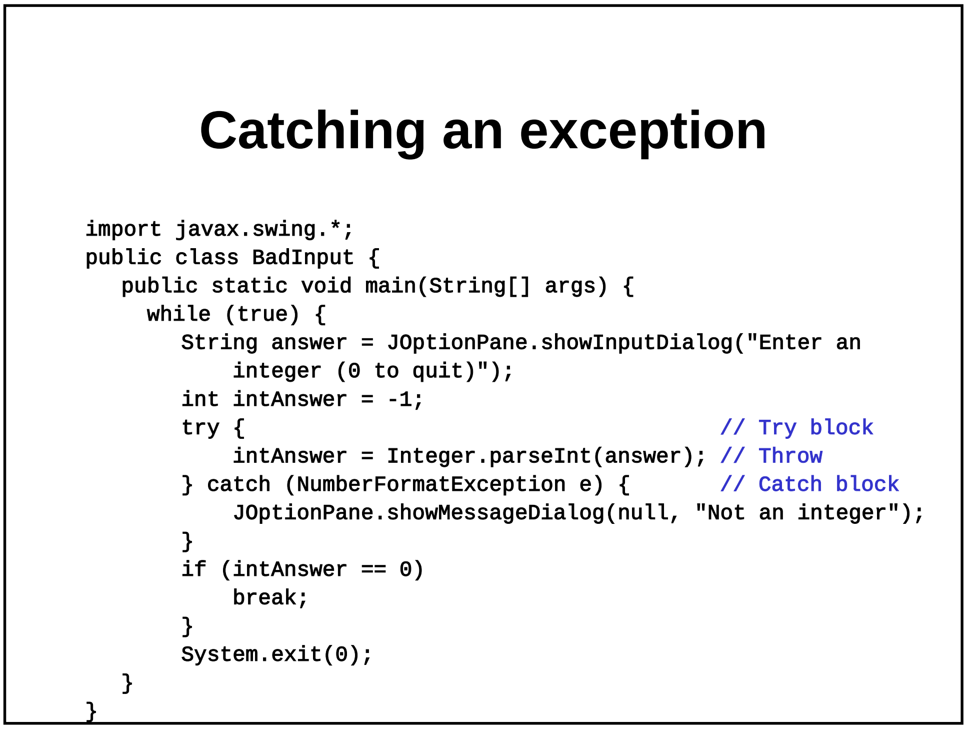
1.00 Lecture 16

Exceptions
Nested and Inner Classes

Reading for next time: Big Java: sections 4.1-4.10

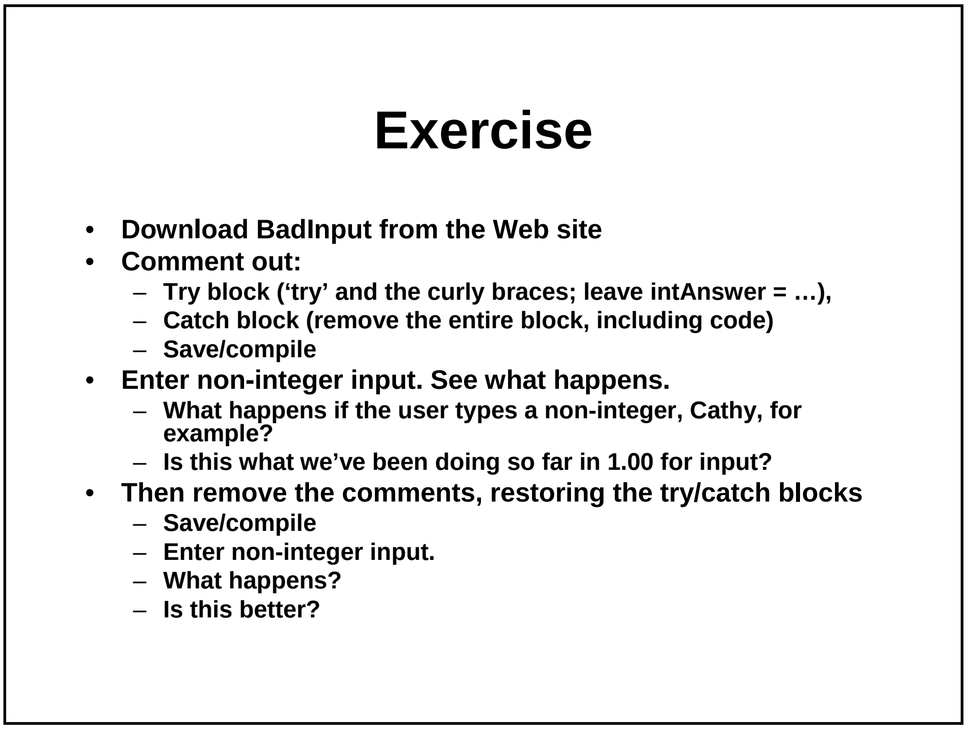
Exceptions: Try, throw, catch

- Exceptions are how Java handles errors that the method in which the error occurs can't handle
- The Java exception mechanism has three elements:
  - Throw (what a method does)
    - Method detects error, cannot handle it
    - Method throws an exception
  - Try block (what the caller of the method does first)
    - Method calls that may throw an exception are placed in a try block (defined by curly braces) in the calling method
  - Catch blocks follow try blocks (what caller does second)
    - Each block contains an exception handler of a given type



Catching an exception

```
import javax.swing.*;
public class BadInput {
   public static void main(String[] args) {
     while (true) {
        String answer = JOptionPane.showInputDialog("Enter an
            integer (0 to quit)");
        int intAnswer = -1;
        try {                                     // Try block
            intAnswer = Integer.parseInt(answer); // Throw
        } catch (NumberFormatException e) {       // Catch block
            JOptionPane.showMessageDialog(null, "Not an integer");
        }
        if (intAnswer == 0)
            break;
        }
        System.exit(0);
   }
}
```



Exercise

- Download BadInput from the Web site
- Comment out:
  - Try block (‘try’ and the curly braces; leave intAnswer = …),
  - Catch block (remove the entire block, including code)
  - Save/compile
- Enter non-integer input. See what happens.
  - What happens if the user types a non-integer, Cathy, for example?
  - Is this what we’ve been doing so far in 1.00 for input?
- Then remove the comments, restoring the try/catch blocks
  - Save/compile
  - Enter non-integer input.
  - What happens?
  - Is this better?

Throwing an Exception

```
public static double average(double[] dArray)
           throws IllegalArgumentException {
    if (dArray.length == 0)
           throw new IllegalArgumentException();
           // Exceptions are objects!
    double sum = 0.0;
    for (int i = 0; i < dArray.length; i++)
           sum += dArray[i];
    return sum / dArray.length;
}
```

Exercise

- **Download class AverageTest, which has:**
  - **The average() method from the previous slide**
  - **A partially written main() method**
    - **Creates nonzero-length array a**
    - **Creates zero length array b as:** `double[] b= { };`
- **Complete the main() method in AverageTest that calls the average() method:**
  - **Call average() twice, with the zero-length and the nonzero-length arrays**
  - **Put the average() calls in a try block and catch the exceptions**
- **Save/compile and run. What happens?**

Exercise

```
public class AverageTest {
   public static double average(double[] dArray)
               throws IllegalArgumentException {
        if (dArray.length == 0)
               throw new IllegalArgumentException();
        double sum = 0.0;
        for (int i = 0; i < dArray.length; i++)
               sum += dArray[i];
        return sum / dArray.length;
   }

   public static void main(String[] args) {
        double[] a = { 1.0, 3.0, 5.0 };
        double[] b = {};
        double avgA = Double.NaN, avgB = Double.NaN;
//      Call average with a and b in try block
//      Catch any exceptions thrown in catch block
        System.out.println("avgA: " + avgA);
        System.out.println("avgB: " + avgB);
   }    }
```

Writing Your Own Exception Classes

- **Writing your own exception class is common.**
- **New exception classes allow you to handle a new type of error separately.**
- **Exception classes extend `Exception`.**

```
public class DataFormatException
  extends Exception {
    public DataFormatException()
      { super(); }
    public DataFormatException(String s)
      { super( s ); }
}
```

Exercise

- Write a ZeroException class that extends Exception
  - Do it exactly as on the previous slide

Exercise, p.2

- Download the ExceptionTest class.
- Write a static quotient(int num, int denom) method that finds num/denom but throws a ZeroException if denom==0
  - Use the constructor for ZeroException that takes a String argument

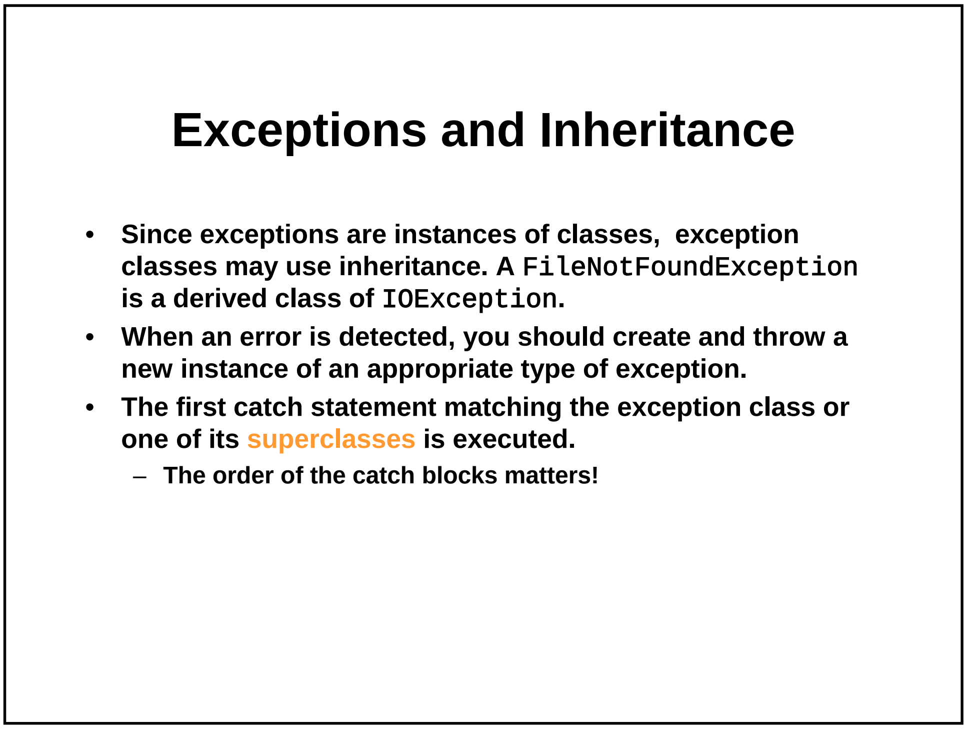
```
import javax.swing.*;
public class ExceptionTest {
    public static double quotient(int num, int denom)
    // Complete the code.
    // Remember method signature must state exception thrown
    // In method body, throw exception when error state occurs
    }

// main() method below
```

Exercise, p.3

- Complete the main() method, partially written for you, to:
  - Read two ints via JOptionPanes (done for you)
  - Call quotient() in a try block and print the result
  - Catch a ZeroException error
  - In the catch block, use:
    - System.out.println(e);   // e is the ZeroException object
  - Do this in a loop that reads ints til –1 is input (done for you)

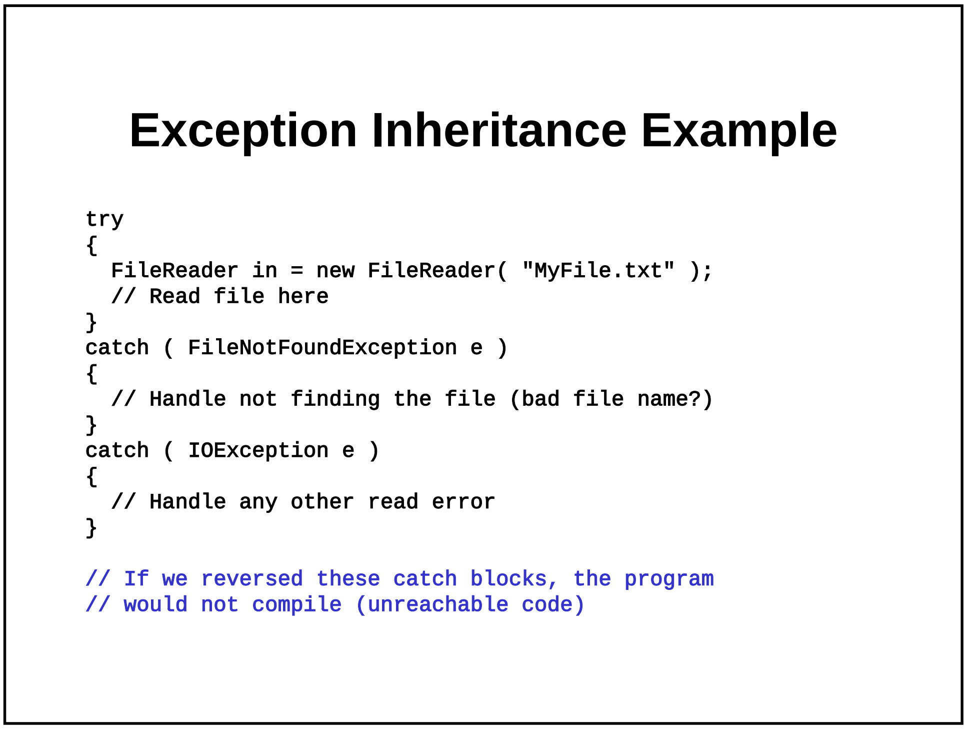
Exercise, p.3

```
import javax.swing.*;
public class ExceptionTest {
    public static void main(String[] args) {
        int num1, num2;
        double result;
        while (true) {
            String answer = JOptionPane.showInputDialog(
                            "Enter an integer" );
            num1 = Integer.parseInt( answer );
            answer = JOptionPane.showInputDialog(
                            "Enter an integer" );
            num2 = Integer.parseInt( answer );
            // Complete the try and catch block code here:
            // Call quotient in try block, catch exception
            if (num1== -1 || num2== -1)
                break; }
        System.exit(0);
    }
}
```



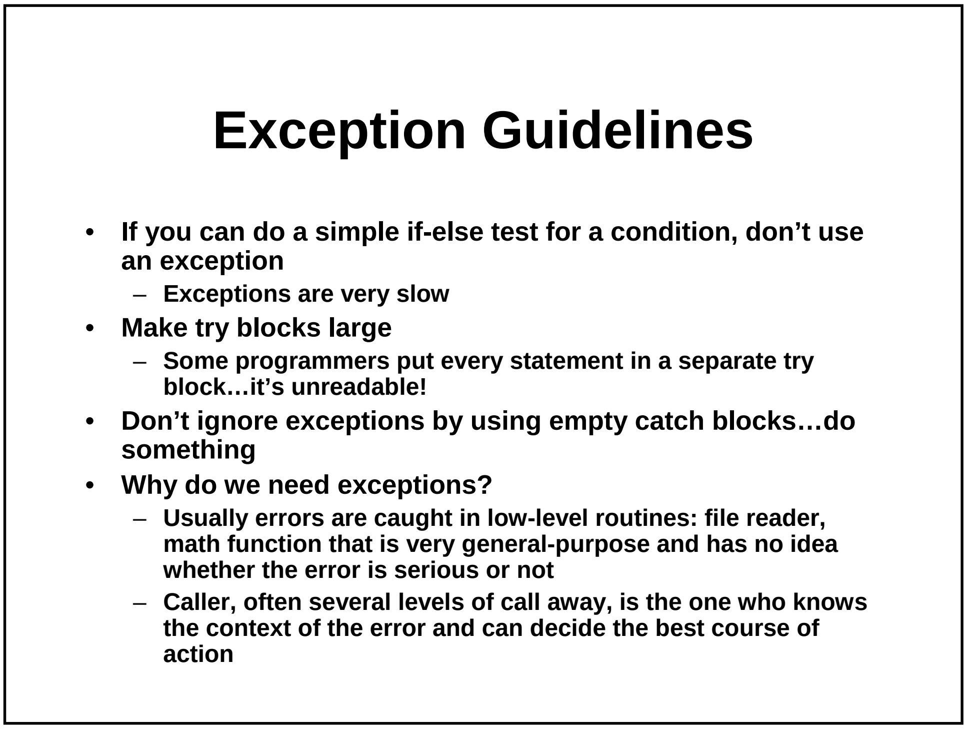
Exceptions and Inheritance

- Since exceptions are instances of classes, exception classes may use inheritance. A `FileNotFoundException` is a derived class of `IOException`.
- When an error is detected, you should create and throw a new instance of an appropriate type of exception.
- The first catch statement matching the exception class or one of its superclasses is executed.
  - The order of the catch blocks matters!



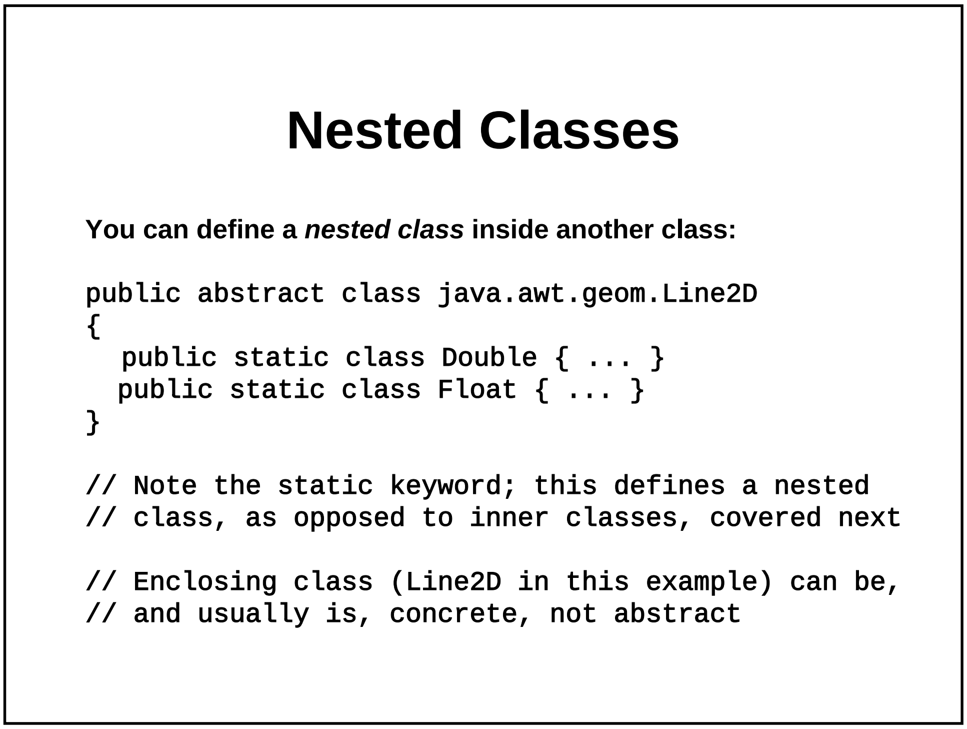
Exception Inheritance Example

```
try
{
  FileReader in = new FileReader( "MyFile.txt" );
  // Read file here
}
catch ( FileNotFoundException e )
{
  // Handle not finding the file (bad file name?)
}
catch ( IOException e )
{
  // Handle any other read error
}

// If we reversed these catch blocks, the program
// would not compile (unreachable code)
```



Exception Guidelines

- If you can do a simple if-else test for a condition, don’t use an exception
  - Exceptions are very slow
- Make try blocks large
  - Some programmers put every statement in a separate try block…it’s unreadable!
- Don’t ignore exceptions by using empty catch blocks…do something
- Why do we need exceptions?
  - Usually errors are caught in low-level routines: file reader, math function that is very general-purpose and has no idea whether the error is serious or not
  - Caller, often several levels of call away, is the one who knows the context of the error and can decide the best course of action



Nested Classes

You can define a *nested class* inside another class:

```
public abstract class java.awt.geom.Line2D
{
  public static class Double { ... }
  public static class Float { ... }
}

// Note the static keyword; this defines a nested
// class, as opposed to inner classes, covered next

// Enclosing class (Line2D in this example) can be,
// and usually is, concrete, not abstract
```

Nested Classes, 2

- Nested class behaves like any other class except that its name is the outer class name concatenated with the inner class name: e.g., `Line2D.Double`
- A nested class is considered to be part of the enclosing class:
  - Make it `public` if you want methods in other classes to use it
  - Make it `private` if you are only going to use it in the enclosing class
- The nested class has no access to the private data (or methods) of the enclosing class, if any
- The enclosing class has full access to all data and methods of the nested class, even if private

Nested Class Example

```
public class Train {
   private int trainNbr;
   private Car[] carList;
   private static class Car {        // Nested class
        private int carNbr;          // Train can access all of it
        private String carType;
        private Car(int c, String ct) { carNbr = c;  carType = ct;}
        private int whatTrain() {return trainNbr; }  // Won't compile
   }
   public Train(int tn, Car[] cl) {
        trainNbr = tn; carList = cl;
   }

   public static void main(String[] args) {
        Car c1 = new Car(5940, "sleeper");
        Car c2 = new Car(5930, "sleeper");
        Car[] cars = { c1, c2 };
        Train t = new Train(59, cars);
        System.out.println(t.carList[0].carNbr + "\n" +
                         t.carList[1].carNbr);   // Private car member
   }
}
```

Inner Classes

- If a nested class is not static, we call it an *inner class*.
- Inner class methods have access to the instance variables and methods of the enclosing class instance.
  - This is the key difference from nested classes
- Why do this?
  - I really don’t know. C++ doesn’t have inner or nested classes. They can be very obscure, so perhaps they’re best avoided.
  - However, they are convenient in Swing; we’ll use them next class as ActionListeners
  - They are regarded by some as a good construct for hiding classes within enclosing classes that are their only user, to prevent any other class from using them
    - However, inner classes can be hacked.

Exercise

- Trains, again
  - We have a train with two cars
    - The train has a number and a voltage at which its power is supplied
    - Each car has a car number and a voltage at which its power operates
  - We want to know for each car:
    - Whether the voltage is compatible
    - Its car number, which is the concatenation of the train number and car number
      - E.g train 59, car 30, yields a car number of 5930
  - Download Train3
    - We’ll complete the code in two steps, on the next slides

Train3 class

```
public class Train3 {
   private int trainNbr;
   private int trainVoltage;              // 480 or 575 volt power
   private Car carA, carB;

   public Train3(int tn, int tv) {
        trainNbr = tn;
        trainVoltage= tv;
   }
   private void setCars(Car c1, Car c2) {
        carA= c1;
        carB= c2;
   }

   public String toString() {
        return ("Train: " + trainNbr +
                "   power compatibility \n  car:" +
                carA.fullCarNbr() + " "+
                carA.isPowerCompatible() + "\n  car:" +
                carB.fullCarNbr() + " "+
                carB.isPowerCompatible());
   }
```

Exercise: Train3 class, contd

```
// Class Car is INSIDE class Train: inner class

private class Car {
     private int carNbr;
     private String carType;
     private int carVoltage;       // 480 or 575 volt power
     private Car(int c, String ct, int v) {
             carNbr = c;
             carType = ct;
             carVoltage= v;
     }
     private String fullCarNbr() {
             // Complete this code: train nbr and car nbr
             return(" ");   // Complete this
     }
     private boolean isPowerCompatible() {
             // Complete this code: train, car voltage same?
             return false;  // Change this to output needed
     }
}
```

Exercise: Train3 main()

```
public static void main(String[] args) {

// Complete main: 5 lines of code.

// 1. Create a new Train, whose number is 59 and voltage is 480.

// 2. Create a new car, number 40, type "sleeper", voltage 480.
//     You need to use odd syntax here:  Car c= t.new Car(...)
//     where t is your train object, because Car is inner class

// 3. Create second car, number 30, type "sleeper", voltage 575.

// 4. Invoke setCars to let the Train know what cars it has.
//     This is odd, but main() created the cars, and Train t
//     doesn't know about them

// 5. Print out the Train t (its toString will do it for you).

}

// Save/compile and run it
```